Student Names: Mohammad Ayub Hanif Saleh (922410888) & Akilan Babu (922453528)
Class: ECS 160
Instructor: Tapti Palit
Date: 02/20/2025

Design Patterns Report

Part 1: Accepting Configuration Options

Requirement:
"Ensure that the application can accept two configuration options—(1) analysis type (weighted, non-weighted), and (2) the name of the JSON file—and ensure that only a single object of this configuration class can exist during the application's lifetime. (Which design pattern should you use?)"

Chosen Pattern: Singleton Pattern

Reasoning:

- We need exactly one instance of the configuration object to exist throughout the application.
- Multiple classes, such as the Analyzer or Visitors, can easily access the configuration (weighted, non-weighted, jsonFilePath).
- A Singleton pattern is perfect because it provides a global point of access and prevents the inadvertent creation of multiple instances.

Implementation:

- We created a class SingleConfig with a private constructor.
- Maintained a static SingleConfig instance and a public getInstance() method.
- The class holds boolean weighted and String jsonFilePath, which are set from command-line arguments as needed.

Part 2: Uniform API for Single Posts/Threads

Requirement:
"Use a design pattern to ensure that both single posts (no replies) and threads (posts with replies) can be handled via the same API. (Which design pattern should you use?)"

Chosen Pattern: Composite

Reasoning:

- A single post is effectively a *leaf* in the Composite structure, while a post with replies is a *composite* node.
- Both leaf and composite implement the same interface (SocialComposite), so client code treats them uniformly.
- The Composite pattern is particularly suited for hierarchical data where items (leaf) and containers (composite) share the same operations.

Implementation:

- Created a SocialComposite interface with methods like:

    - get_post_Id(), get_post_replies(), add_reply_under_post(), accept(visitor)
- A Post class implements SocialComposite.
- A post with an empty list of replies behaves like a leaf, while a post with child replies behaves like a composite. The accept(visitor) method loops over all replies, if they exist.

Part 3: Computing Statistics via Visitors

Requirement:
"Use design patterns to compute the statistics. These design pattern classes should visit each post and reply for all top-level posts and threads in input.json and compute the required statistic. (Which design pattern should you use? How many classes did you create?)"

Chosen Pattern: Visitor

Reasoning:

- This pattern separates analytic/business logic (counting, averages, weighted analyses) from the Post class.
- Each post calls visitor.visit(this), and the composite structure recurses through replies automatically.
- Makes it easy to add new analytics by adding new `Visitor` classes. Additional analytics functions should implement the SocialVisitor interface.

Implementation:

- Post.accept(visitor) calls visitor.visit(this) and then calls accept on each reply in its list.
- Each concrete Visitor implements a visit(Post post) method and accumulates results in local fields.
    - We use CountingVisitor, ReplyVisitor, and AverageDurationVisitor classes to calculate the total posts, average replies, and average reply duration, respectively.
- After traversal, we retrieve the results (countingVisitor.getCount()) and display or store them.

Part 4: Adding Hashtag Functionality Without Modifying or Subclassing Post

Requirement:
"Extend HW1 to add the hashtagging functionality from HW3, storing the hashtag in the post/thread object without adding a hashtag field to Post or subclassing it. (Which design pattern should you use?) Then print hashtags for all posts and threads, including those without one."

Chosen Pattern: Decorator

Reasoning:

- We want to attach extra data (hashtag) to Post objects without altering the Post class itself or creating a subclass.
- The decorator wraps an existing object, implementing the same interface while adding new behavior (the hashtag).
- This preserves the original class design but adds new functionality.

Implementation:

- A HashtagDecorator class implements SocialComposite and stores a reference to the wrapped Post.
- It decorates each post (and optionally replies) with a String hashtag.
- By overriding get_post_replies() to return decorated children, each reply is also decorated and can have its hashtag.
- In the final logic (SocialAnalyzerDriver), we:
    - Sort the top-level posts by like_count.
    - Select the top 10.
    - Wrap each with HashtagDecorator, which internally calls a local LlamaInstance.generateHashtag(...) to populate the hashtag.
    - Print out each post's content + hashtag, along with some replies.

Summary

1. **Singleton Pattern**: SingleConfig is a configuration object that can adjust the weighted vs. non-weighted analysis and JSON file path. Only one instance of this object exists.
2. **Composite Pattern**: Post implements SocialComposite, treating single posts as leaves and posts with children as composites.
3. **Visitor Pattern**: Classes like CountingVisitor and ReplyVisitor compute statistics (counts, averages, durations) without modifying the Post class.
4. **Decorator Pattern**: HashtagDecorator adds hashtag functionality without changing or subclassing Post. We recursively wrap replies so each child can also have a hashtag.

In the end, we updated SocialAnalyzerDriver.java to demonstrate how we sort posts by like_count, take the top 10, and decorate them. We use HashtagDecorator to generate hashtags (via the local LLaMA-based generator LlamaInstance), and we show 2 replies with hashtags as well.